SecDINT: Preventing Data-oriented Attacks via Intel SGX Escorted Data Integrity

Dakun Shen*, Tao Hou†, Zhuo Lu‡, Yao Liu‡, and Tao Wang§
*Zhejiang Lab, Hangzhou, ZJ, PRC, shendakun@zhejianglab.com
†Texas State University, San Marcos, TX, USA, taohou@txstate.edu
‡University of South Florida, Tampa, FL, USA, {zhuolu@, yliu@cse.}usf.edu
§University of North Carolina at Charlotte, Charlotte, NC, USA, twang27@uncc.edu

***Abstract*—Data-oriented attacks with the intent to corrupt critical memory data without violating Control-flow Integrity (CFI) pose significant threats to legitimate program execution. Existing mitigations predominantly rely on software-based memory safety measures to ensure critical data integrity, a solution often associated with elevated performance overhead and susceptibility to intricate attack techniques. In this paper, we present a CPU level data integrity design, named Intel SGX Escorted Data Integrity (SecDINT), to automatically protect sensitive variables against data-oriented attacks. Our design can achieve the data integrity of sensitive variables via SGX enforced isolation in binaries. We evaluate SecDINT on real-world applications. The results reveal that SecDINT can effectively identify sensitive variables, enforce data integrity, and prevent data-oriented attacks. Comparative analysis with existing software-based strategies (e.g., 103% runtime overhead in Data-flow Integrity, 116% in SoftBound with CETS), showcased SecDINT's remarkable capability in drastically reducing overhead to as low as 20.1%.**

***Index Terms*—Data-oriented Attacks, Data Integrity, Data-flow Integrity, Intel SGX, Data Enclave**

I. Introduction

With the wide deployment of Control-flow Integrity (CFI), traditional memory corruption attacks aimed at hijack the control flow have become increasingly difficult. Nevertheless, attackers continue to seek alternative ways to exploit memory vulnerabilities to achieve malicious goals. Consequently, data-oriented attacks [16] have emerged. These attacks aim to manipulate non-control data (e.g., a data variable or pointer that does not contain the target address for a control transfer) to subvert a program's normal execution. Data-oriented attacks can be equally powerful and effective as control-flow attacks. The attacker is able to tamper with application-specific data to cause significant damage, such as information leakage [9], [15], user privilege escalation [12], [16], and arbitrary code execution [9]. Remarkably, the exploited program could still follow the Control-flow Graph (CFG), which makes the attack difficult to combat.

Develop effective defense strategies that can contain data-oriented attacks is urgent. These strategies should be tailored to guarantee the data integrity of sensitive variables in a vulnerable program's memory. Recent developments of such defenses include Data-flow Integrity (DFI) [6], SoftBound [21], YARRA [29], HardScope [25], and PrivWatcher [8]. However, these defense strategies meet major limitations:

- These existing strategies heavily rely on software-based mechanisms (e.g., Shadow Stack [3], SafeStack [7], and Safe-Region [8]) to enforce memory isolation and data integrity. They protect the sensitive data from being altered by limiting write access or storing a verifiable replica. Notably, these strategies either suffer from high performance overhead without hardware acceleration or can only be deployed in very limited usage scenarios [10]. More importantly, these strategies can be circumvented by advanced attacks due to the inherent vulnerabilities of software-based mechanisms [10].
- Identifying the sensitive variables vulnerable to data-oriented attacks is difficult. Usually, this identification requires specific domain knowledge and must be done manually. This may impose a significant burden for the deployment of these strategies. Additionally, most of these approaches cannot work for COTS binaries, while requiring performing on the source code of a program.

These drawbacks motivate us to develop a more appropriate defense strategy from the hardware perspective. Towards this, we propose a practical system, named SGX Escorted Data Integrity (SecDINT), to defend against data-oriented attacks. Specifically, SecDINT applies Intel SGX technology [18] to create a memory enclave in such a way that data-oriented attacks cannot use traditional memory corruption methods to tamper with sensitive variables. It can automatically identify sensitive data in a given binary and move such data into the SGX enclave for protection. SecDINT also aims to optimize performance, moving only the most crucial variables to the SGX enclave to reduce overhead. SecDINT offers these advantages:

- Lightweight: SecDINT can significantly reduce the performance overhead when enforcing data integrity with the assistance of Intel SGX.
- Ubiquitous: SecDINT can be deployed to any systems equipped with Intel SGX enabled processors.
- Automatic: SecDINT is an automatic approach to enforce data integrity without requiring domain knowledge.
- Source-Code-Free: SecDINT can be performed on binaries to protect sensitive variables from being corrupted.

SecDINT is composed of three modules: (i) the analysis

module, (ii) the sensitive variable identification module, and (iii) the instrumentation module. To work, SECDINT first analyzes a target binary and constructs its basic blocks, Control-flow Graph (CFG), and Data-flow Graph (DFG) in the analysis module. Then, the identification module will identify sensitive variables that are vulnerable to data-oriented attacks. Finally, the instrumentation module instruments the target binary to protect the identified sensitive variables using Intel SGX. The instructions for enclave-related operations are also instrumented in this step.

In the implementation of SECDINT, we encounter the following challenges and propose corresponding techniques to address them.

(1) It's challenging to effectively identify the sensitive variables. To address this challenge, we comprehensively investigate data-oriented attacks [12], [9], [16], [5], [17] to find the key factors contributing to the success of such exploits. We find that three types of variables are sensitive: i) the conditional branching data, ii) the arguments that will be passed to system or library function calls, and iii) their dependent data. By protecting these three types of sensitive variables from corruption, the threat of data-oriented attacks can be significantly contained. Furthermore, SECDINT is designed to be customizable, allowing the system can be extended by adding more types of sensitive variables based on needs.

(2) It's non-trivial to optimize the performance of SECDINT. Our goal is to reduce the number of sensitive variables to be protected while maintaining the same level of security assurance. We have designed an optimization method that can further identify the crucial sensitive variables originating from or arithmetically derived from untrusted sources (e.g., network sockets and unprivileged users). To achieve this purpose, SECDINT integrates dynamic taint analysis [23] to identify all variables that originate from untrusted sources.

We aim to implement SECDINT as a practical system. In addition to the aforementioned novel designs, we have also meticulously chosen multiple state-of-the-art binary analysis techniques to achieve the best possible development outcomes. In the analysis module, we utilize ANGR [30] to extract the CFG and DFG, while employing INTEL PIN [31] for dynamic taint analysis. In the instrumentation module, we make use of UROBOROS [32] to disassemble stripped binaries and generate reassemble-able assembly code. During the evaluations of SECDINT, we conduct testing on a range of real-world applications. The results demonstrate that SECDINT effectively identifies sensitive variables, enforces data integrity, and thwarts data-oriented attacks. Additionally, we assess the performance of SECDINT. In comparison to existing software-based strategies (e.g., 103% run-time overhead in data-flow integrity [6], 116% in SoftBound with CETS [21]), SECDINT substantially reduce the overhead to as low as 20.1%.

II. MOTIVATION AND SYSTEM OVERVIEW

In this section, we provide a brief introduction to the background and our motivation for protecting sensitive variables to defend against data-oriented attacks.We then present an overview of SECDINT.

A. Data-oriented Attacks

Traditional control data attacks manipulate a program's control data (e.g., return addresses or function pointers) to hijack or redirect the target program's control flow [9]. In contrast, data-oriented attacks corrupt only data, without affecting code pointers. These attacks also follow legitimate paths on the program's CFG, rendering existing control-related defenses ineffective [17]. Nonetheless, data-oriented attacks can be as powerful and effective as control-flow attacks, resulting in significant damages, including information leakage [9], user privilege escalation [12], [16], and arbitrary code execution [9].

Code 1 provides a typical example of data-oriented attacks. The code is abstracted from a vulnerability in SSH [33] that can be exploited to launch a user privilege escalation attack. In this example, the `HandlePacket` function on line 4 does not validate the `packet`'s length and can be manipulated to write over 1000 bytes to `packet` when receiving input from the network. By utilizing this exploit, an attacker can overwrite the `authenticated` variable, circumventing the conditional check on line 5 and gaining access to the processed `packet`.

```
1 int authenticated = 0;
2 char packet[1000];
3 while(!authenticated)
4   authenticated = HandlePacket(packet); //buffer
      overflow
5 if(authenticated)
6   ProcessPacket(packet)
```

Code 1. Vulnerable code snippet.

B. Intel SGX Escorted Data Integrity

To mitigate data-oriented attacks, various software-level defenses have been developed [6][21][29][25][8]. However, these defenses mechanisms often necessitate comprehensive software-level memory safety enforcement to ensure the integrity of sensitive data, resulting in a high performance overhead. Moreover, relying solely on software-based isolation strategies can render systems vulnerable to advanced attacks, as these approaches are susceptible to exploits and breaches inherent to software vulnerabilities [10]. Additionally, they require knowledge of a program's source code to implement protection against data-oriented attacks. These challenges drive our motivation to design an effective protection of sensitive variables, focusing on defending against data-oriented attacks from a hardware perspective.

Intel SGX offers a set of CPU instructions capable of establishing isolated regions (enclaves) for code and data within user applications. This feature makes it an ideal solution for ensuring the data integrity of sensitive variables through hardware-level memory isolation. Our objective is to automatically instrument SGX code into binaries for safeguarding sensitive variables against data-oriented attacks. With this goal in mind, we propose SECDINT.

To circumvent the authentication and manipulate the packet processing in Code 1 , an attacker can exploit the buffer overflow vulnerability in `HandlePacket` to tamper with the `authenticated` variable. The corresponding assembly code for Code 1 is provided in Code 2, indicating that the `authenticated` variable resides at `-0x3f4(%rbp)`, and the `packet` variable begins at `-0x3f0(%rbp)` within the binary. If we can identify `-0x3f4(%rbp)` as conditional branching data, we can secure it from corruption by transferring this variable into the SGX enclave. Additionally, we must modify the read/write instructions related to `-0x3f4(%rbp)`, such as lines 1 and 12 in Code 2, to ensure the correct execution of the binary program.

```
1  movl $0x0,-0x3f4(%rbp)     #authenticated = 0
2  jmp  .BB1                  #jump to while_loop
3  .BB2:
4  movl -0x3f0(%rbp),%rax     #move packet to %rax
5  movq %rax, %rdi
6  callq HandlePacket         #buffer overflow
7  movl %eax,-0x3f4(%rbp)     #result -> authenticated
8  .BB1:
9  cmpl $0x0,-0x3f4(%rbp)     #check authenticated
10 je .BB2                    #jump to while_loop
11 cmpl $0x0,-0x3f4(%rbp)     #check authenticated
12 je .BB3
13 movl $0x0,%eax
14 callq ProcessPacket        #call ProcessPacket()
15 .BB3:
16 movl $0x0,%eax
```

Code 2. Vulnerable assembly code snippet.

C. Threat Model

We establish a threat model that involves a potent adversary possessing full control over the data memory within the target program. This encompassing scenarios includes buffer overflows and other memory corruption vulnerabilities, which have the potential to result in data memory corruption. Additionally, state-of-the-art mechanisms, such as CFI [1], DEP [20], and ASLR [26], have been adopted to effectively defend against control-data attacks. Our adversary model is in line with conventional run-time attack scenarios and resonates with the framework elucidated by Hu *et al.* in their analysis of DOP attacks [17]. Also, side-channel attacks against Intel SGX are out of the scope of this paper.

D. SECDINT

SECDINT is an automated tool designed to search for sensitive variables within binaries and instrument SGX-enabled safeguards for such data. The architecture of SECDINT is illustrated in Figure 1. This framework comprises three main modules in SECDINT: i) the analysis module, ii) the sensitive variable identification module, and iii) the instrumentation module.

As shown in Figure 1, SECDINT initially analyzes a target binary, constructing its basic blocks, CFG and DFG within the analysis module. Subsequently, the identification module identifies sensitive variables that are susceptible to data-oriented attacks. Moving forward, the instrumentation module instruments the target binary by relocating the identified data into the SGX enclave, thereby ensuring protection. Furthermore, this module creates the enclave-related code to store and operate the sensitive variables.

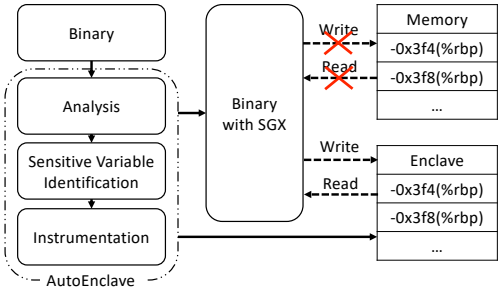

Fig. 1. System architecture of SECDINT.

III. SENSITIVE VARIABLE IDENTIFICATION

The analysis module and the identification module within SECDINT function to analyze the assembly code of a binary, facilitating the identification of sensitive variables. In this section, we present the algorithms employed for sensitive variable identification.

A. Sensitive Variables

It is necessary to identify sensitive variables that are vulnerable to memory corruption attacks within binaries. In order to limit the cost of communications between the binary and the enclave, the set of sensitive variables should be kept minimal while encompassing all data that is vulnerable to data-oriented attacks. Based on our investigation of exiting data-oriented attacks in literatures [12], [9], [16], [5], [17], [2], we identify three types of sensitive variables: i) conditional branching data, ii) parameters passed to system and library function calls, and iii) their dependent data.

1) Conditional branching data: Branch instructions are treated as a lifeline of a program, as they decide the program execution path according to the variety of conditional branching data. One of the most powerful data-oriented attacks is designed to corrupt conditional branching data or variables associated with them. This manipulation redirects the program's control flow towards arbitrary locations. Code 1 is an example of a privilege escalation attack, wherein the conditional variable `authenticate` is tampered to grant authentication for the received packet. Thus, isolating such data in a more secure location will significantly reduce the risk of data-oriented attacks.

```
1 struct passwd { uid_t pw_uid; ...} *pw; ...
2 int uid = getuid();
3 pw->pw_uid = uid;
4 ...                    //format string error
5 void passive (void) {
6   ...
7   setuid(0);
8   setuid(pw->pw_uid);
9   ...}
```

Code 3. An example of data-oriented attacks.

2) Parameters passed to system and library function calls: Another type of sensitive variables is parameters passed to system and library function calls. System calls like `setuid` and `sys_fchown`, governed by the kernel, manage system and hardware resources. Similarly, library function calls like `printf` and `strcpy` are also indispensable to process necessary demands of both applications and users. Corrupting the content of those arguments can lead to serious damage, such as privilege escalation and information leakage. Code 3 demonstrates a working exploit fashioned after the WU-FTPD server code [9]. In the exploit, by making use of a format string error, an attacker tampers the parameter `pw->pw_uid` within the `setuid` system call, resulting in the elevation of user privilege.

3) Dependent data: Only isolating conditional branching data and function arguments is not enough to guarantee the integrity of sensitive variables. Consider Code 3 as an example: the variable `pw->pw_uid` is passed as a parameter to the system function `setuid()` on line 10. Before variable `pw->pw_uid` is used, it is assigned by the variable `uid` on line 4. Thus, `uid` is the dependent data for `pw->pw_uid`, as the value of `pw->pw_uid` is directly related to `uid`. An attacker can corrupt the dependent variable `uid` to indirectly alter the sensitive variable `pw->pw_uid`. Therefore, in order to guarantee the integrity of conditional branching data and critical function parameters, it is essential to extend protection to all data upon which they depend.

B. Sensitive Variable Identification

Schemes utilizing source code have inherent advantages in correctly identifying vulnerable variables due to the richer context they provide. In contrast, when performing analysis on binaries, variables are denoted by an address expression such as $[base + index \times scale + offset]$, which makes them challenging to identify.

To solve this issue, we have designed an identification method aimed at automatically locating and designating the three types of sensitive variables within binaries. We assume that given a target binary file f, the analysis module in SECDINT generates a set of basic blocks denoted by B, a CFG denoted by G, and a DFG denoted by D. A basic block $b \in B$ consists of a sequence of instructions with no branches except in the entry or the exit. By taking B and D as inputs, the identification method produces a set of locations of sensitive variables S in the target binary file as an output. The identification method is constructed by multiple parts, which are demonstrated in the following.

a) Identify conditional branching data.: The basic idea is to check each basic block and identify data associated with comparison and branch instructions. Specially, from the given set of basic blocks B, SECDINT first locates the branch instructions, such as `je` and `jne`, in each basic block b in set B. If a branch instruction is found in the basic block b, $currentInt$ is pointed to the conditional instruction `cmp` that is associated with the branch instruction. Then the operation instruction $currentInt$ is analyzed to check whether the operand of this instruction is a variable, such as `%rbp` and `%rsp`. If the operand is a variable denoted by d, the memory location of d is recorded in the output set S. If the operand is a regular register, which indicates the value in the register is passed from previous instructions, SECDINT recursively analyzes each previous instruction until reaching the beginning of the function or encounters an instruction that assigns the register from a variable.

b) Identify parameters in system and library function calls.: To identify the parameters, our method first locates the `call` operation in each basic block and then identifies all corresponding data according to the operation's calling convention. Our approach focuses on protecting Linux x86-64 binaries with the calling convention that is specified by the System V ABI. In this architecture, parameters of the integer, boolean, enumerated, or pointer type in a function call are passed in six registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9` in order. Float and double types are transferred in `xmm0`, `xmm1`, `xmm2`, `xmm3`, `xmm4`, `xmm5`, `xmm6`, and `xmm7` registers in order. Additional parameters are stored on the stack [13]. When identifying the parameters, our approach first identifies a system or library function call. Next, by matching the type, the number, and the order of the data of this function to the corresponding positions in the calling convention, our approach extracts the parameters in the specific function call.

During the identification process, one issue to be noted is how to handle indirect calls such as function pointers. A function can be called indirectly through a register instead of an identifier, thus it is difficult to identify which function is being called by static analysis. To address this issue, our approach recovers the procedure linkage table (PLT) from the target binary. The PLT contains absolute addresses for all system and library function calls that are involved in the target binary. Therefore, by comparing the calling convention of the function pointer with each calling convention of the function in the PLT, we can identify which function is being called indirectly. If a function pointer corresponds to one or more functions within the PLT, the parameters linked to this function pointer are deemed sensitive variables in our methodology. In cases where a function pointer does not align with any function within the PLT, our approach interprets this function as a user-defined function and disregards it.

c) Identify dependent data.: To identify the dependent data, SECDINT recursively extract all dependent data by analyzing the context-sensitive DFG [30]. Specifically, SECDINT recursively takes each variable k in set S as an input and outputs the updated set S with all data that k depends on. We first examine whether k has been recorded before. If it is not in set S, SECDINT updates set S with variable k, which is further passed as a parameter to function `processDFG`. Function `processDFG` analyzes the DFG and outputs all data that k depends on into set U. Next, each variable in set U is examined by recursive manner. If the data in set U is never recorded, it will be analyzed recursively. Therefore, all dependent data associated with conditional branching data and critical function parameters can be identified and recorded.

IV. SecDINT Enforcement via Binary Instrumentation

With all sensitive variables identified, the instrumentation module of SecDINT automatically instruments the binary to protect these sensitive variables. In this section, we present the details of the binary instrumentation component module in SecDINT.

A. Binary Instrumentation

The binary instrumentation module in our approach rewrites binaries in a way that they are forced to write and read sensitive variables to/from the protected enclave memory. We adopt Uroboros [32] as the tool for binary instrumentation. Uroboros can automatically disassemble stripped binaries and generate reassemble-able assembly code, which makes it suitable to our approach. By adopting Uroboros, we are able to directly work on the assembly code of binaries to add Intel SGX instructions and modify all operations associated with sensitive variables. Moreover, since the assembly code extracted from binaries can be reassembled back to executables, it is convenient to link binaries with the Intel SGX libraries during the reassembly process [14].

Figure 2 shows the overall binary instrumentation design. The left-hand side of this figure is the architecture of an ELF binary file. The right-hand side shows what information and which part of the binary file are instrumented. Particularly, three sections in the binary file need to be modified. The `.text` section houses all major functions for the enclave operations, such as enclave initialization and data transaction. The `.bss` section and the `.rodata` section hold the enclave parameters, such as global variables and local variables.

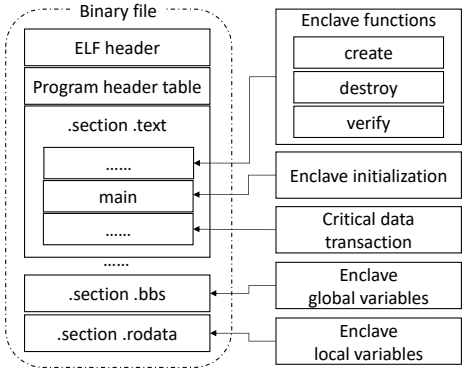

Fig. 2. Binary instrumentation.

The `.text` section contains executable code. Therefore, additional functions need to be instrumented into this section in order to access to the enclave memory from the executable. In this procedure, the first step is to initialize the enclave at the beginning of the execution. Intel demonstrates how to write the enclave initialization function in C++. Since our approach directly works on assembly code, we translate the initialization function to assembly code and add it to the `.text` section as a regular function. A function call is added at the beginning of the execution so as to call the initialization function. Second, all instructions which operate sensitive variables need to be instrumented. Specificlly, all write and read operations associated with sensitive variables are replaced with corresponding enclave functions. These functions call the enclave functions that are defined in the `enclave.cpp` file to write or read sensitive variables to/from the enclave. Third, global and local variables, which are used by the enclave, are instrumented into the `.bss` section and `.rodata` section, respectively. The global variables include a global enclave ID and a global enclave handle pointer. These two variables are used to communicate with the enclave. The local variables contain the enclave debug information.

B. Sensitive Variable Type Construction

Our approach moves all sensitive variables from binaries into enclave memory to gain a high-level protection. Intel enclaves are written in C/C++ code, therefore types of sensitive variables need to be recovered in order to correctly initialize and store them in the enclave memory. However, re-constructing data type in a binary without source code is not straightforward, because high-level data types are typically stripped by the compiler during the compilation process. Since all data is decoded into binary code represented by 1 and 0 when they are processed in registers at the lowest level of a machine, all sensitive variables can be treated as integers (at C/C++ code level) in our approach when creating enclave memory for them.

C. Variables on the Heap

The size of a variable allocated on the heap can be arbitrary, since it may be created at run time. This makes it difficult to get the correct size of the blocks on the heap. To address this issue, instead of moving the entire buffer allocated on the heap into the enclave, we first record all variables allocated on the heap with an index. When this variable is written into the heap, the binary value of this variable is also stored into the enclave. Correspondingly, when this variable is read, other than reading it from the heap, the process is forced to load the variable from the enclave by matching its index.

Considering Code 4 as an instance, part (a) shows that a variable is allocated on the heap with size `0x400` by function `malloc`. A pointer `-0x8(%rbp)` is used to point to this variable, which is later assigned by the library function `fread()`. Part (b) is the instrumentation code corresponding to part (a). After the variable on the heap is assigned on line 10, this variable is also stored into the enclave for protection. Part (c) shows that the same variable is read from the heap by using pointer `-0x8(%rbp)` and is further passed into `printf` function. Part (d) demonstrates the corresponding instrumentation code for Part (c). Instead of loading data from the heap, an effective address in register `%rax` is copied in the pointer `-0x8(%rbp)` by instruction `leaq` on line 4. The effective address in register `%rax` is further used to point to the value loaded from the function `enreadV9`, which loads variable from the isolated enclave to the outside.

```
1  ...
2  mov $0x400,%edi
3  callq malloc
4  mov %rax,-0x8(%rbp)
5  ...
6  mov -0x8(%rbp),%rax
7  mov %rdx,%rcx
8  ...
9  mov %rax,%rdi
10 callq fread
11 ...
12
13
14
15
16
17
18
```
(a) write to the heap

```
1  ...
2  mov $0x400,%edi
3  callq malloc
4  mov %rax,-0x8(%rbp)
5  ...
6  mov -0x8(%rbp),%rax
7  mov %rdx,%rcx
8  ...
9  mov %rax,%rdi
10 callq fread
11 #Save states
12 mov -0x8(%rbp),%rax
13 mov (%rax),%rdx
14 leaq gloedi(%rip,%rax)
15 movl %edx,%esi
16 movl %rax,%rdi
17 callq enSaveV9
18 #restore states
```
(b) write to the enclave

```
1
2
3
4  ...
5  mov -0x8(%rbp),%rax
6  mov %rax,%rsi
7  mov $S_0x400,%edi
8  mov $0x0,%eax
9  callq printf
10 ...
11
12
13
14
15
16
17
18
```
(c) read from the heap

```
1  ...
2  #Save states
3  #mov -0x8(%rbp),%rcx
4  leaq -0x8(%rbp),%rax
5  movq %rax,%rcx
6  leaq gloedi(%rip),%rax
7  movq (%rax),%rax
8  movq %rcx,%rdx
9  movq %rdx,%rsi
10 movq %rax,%rdi
11 callq enReadV9
12 #restore states
13 leaq -0x8(%rbp),%rax
14 mov %rax,%rsi
15 mov $S_0x400,%edi
16 mov $0x0,%eax
17 callq printf
18 ...
```
(d) read from the enclave

Code 4. An example of the instrumentation method for protecting critical data transactions from the heap. Sensitive variables are marked in red.

V. Performance Optimization

SecDINT remains functional even when tasked with protecting a comprehensive range of variables within a program. However, it's essential to consider the trade-off introduced by the fact that accessing data in enclave memory is significantly slower – approximately 300 times – compared to accessing data in regular memory. To optimize the performance of SecDINT, an intuitive approach involves the further refinement of sensitive variables, thereby reducing the size of the enclave. To achieve this optimization, we have designed a method that exclusively isolates imperative sensitive variables that are influenced by data originating from or derived through arithmetic operations involving untrusted sources – comprising network sockets and unprivileged users. In pursuit of this objective, we apply dynamic taint analysis [23] to label all data originating from network sockets and unprivileged users (i.e., from stdin or files), and track program execution to detect the propagation of these tainted data. Specifically, to perform dynamic taint analysis, we created a Intel Pin tool [31] that instruments the binary to track the flow of untrusted sources as it moves through the program. The tool identifies sources of tainted data, propagates the taint information through instructions, and analyzes how it interacts with program execution. The tool generates outputs or logs indicating where tainted data goes and how it affects the program. This process is conducted separately from the enclave environment and is removed upon completion. Consequently, this process does not compromise enclave security.

Figure 3 demonstrates an example of this optimization process. It shows a partial CFG generated from Code 2. The `-0x3f0(%rbp)` variable in the BB_3 block is tainted as it comes from a network socket. Therefore, the BB_3 and its parent blocks, BB_0 and BB_2, are identified as the field that are under the influence of an untrusted source. Sensitive variables on these blocks should be isolated for protection. In this case, the `-0x3f4(%rbp)` and `-0x18(%rbp)` variables are selected. Other sensitive variables, such as the `-0x10(%rbp)` variable in the BB_5 block, are excluded.

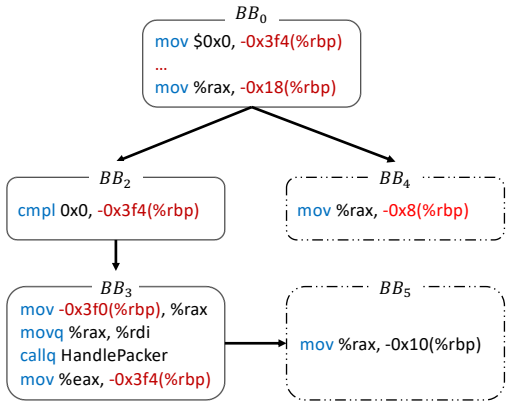

Fig. 3. Sensitive variable refinement. Sensitive variables are marked in red. Solid boxes represent the blocks under the influence of untrusted sources.

VI. Evaluations

We prototype SecDINT for x86-64 ELF binaries on the Ubuntu 16.04 64-bit system. Multiple state-of-the-art binary analysis techniques are integrated in our approach. In the analysis module, we adopt angr [30] to provide the CFG and DFG constructions and Intel Pin [31] to taint sensitive variables. In the instrumentation modules, Uroboros [32] is employed to disassemble stripped binaries and generate reassemble-able assembly code. By combining objdump and Uroboros, our approach is able to analyze a target binary and provides an instrumented assembly code with Intel SGX enabled which can be further reassembled back to the binary.

In this section, we conducted an evaluation of SecDINT focusing on the accuracy of sensitive variable identification , the efficiency of instrumentation, and its efficacy in countering data-oriented attacks. Our evaluation used two collections of binaries: (1) the GNU core utilities, referred as Coreutils,

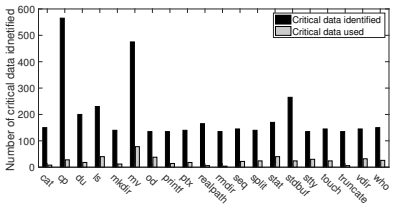

Fig. 4. COREUTILS identified sensitive variables.

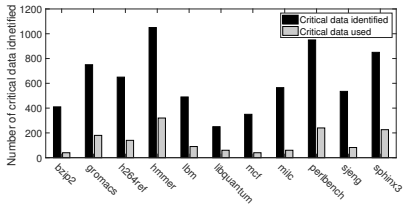

Fig. 5. SPEC identified sensitive variables.

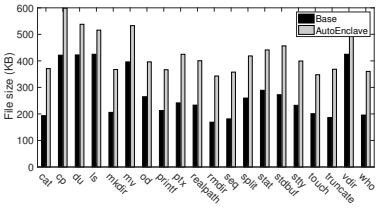

Fig. 6. COREUTILS space overhead.

including 100 binary utilities for basic file, shell and text manipulation. (2) the C benchmarks from the SPEC CPU benchmark package [11]. Furthermore, we implemented multiple data-oriented attacks to evaluate the protection ability of SECDINT.

A. Sensitive Variable Identification Correctness

To measure the correctness of the sensitive variable identification, we compared the sensitive variables identified by SECDINT with the actual sensitive variables during execution. Specifically, we use Intel Pin [19] to dynamically capture the sensitive variables during execution. The results are shown in Figure 4 and Figure 5. A key observation from these two figures is that the number of sensitive variables used during execution is significantly smaller than the number of sensitive variables identified by our method. A comparison result shows that all sensitive variables and dependent data used during execution is identified by our identification method, which indicates that our sensitive variable identification method may over-identify, but not lose any sensitive variables. This ensures SECDINT can provide effective protection against data-oriented attacks.

B. Binary Instrumentation Efficiency

1) File Size Overhead: We compare the file size of instrumented binaries with the original binaries. The results are whown in Figures 6 and 7. The average space overhead is 39.7% for COREUTILS and 4.1% for SPEC CPU. This result meets our exception that the overhead for COREUTILS is larger than SPEC CPU. The resason is that the original COREUTILS binaries are relatively smaller than the SPEC CPU binaries, while the total size of the enclave codes is roughly one third of the size of the COREUTILS. For instance, the original `cat` binary has 14938 lines of code. After instrumentation, it increases to 19134 lines of code.

2) Run-time Execution Overhead: To run-time overhead is mearsured with the SPEC CPU benchmarks. The results are shown in Figure 8 with an average execution overhead of SPEC is 128%. To better understand the execution time overhead of our approach, we also measured the execution times of accessing data in the ordinary memory (i.e., memory outside the enclave) and the enclave. We designed a program that repeatedly reads/writes data from/to the ordinary memory and the enclave, and calculated the average execution time for each operation. Results show that accessing data in the enclave is around 300 times slower than accessing data in the ordinary memory, which indicates that when enforcing SECDINT, more sensitive variables to be protected can lead to more execution time overhead. Therefore, the set of sensitive variables should be as small as possible in order to minimize the performance overhead.

C. Optimization Improvement

In this section, we describe the results of the optimization design. As disscussed, refining sensitive variables will lead to a significant performance improvement. To evaluate effectiveness of this optimization design, we compare the run-time overheads for with and without optimization. The results are shown in Figure 9, which indicates that the optimization method can achieve significant performacne improvement for SECDINT. The average run-time execution overhead is reduced by 56.7%. The results also show that, with the optimization, the run-time overhead can be significantly reduced to as low as 20.1%.

D. Mitigating Attacks

To evaluate if SECDINT is able to prevent data-oriented attacks, we implemented multiple real-world data-oriented attacks and tested our design with them.

1) ***GHTTPD***: GHTTPD is a lightweight HTTP server. A buffer overflow vulnerability in the `Log` function allows attackers to execute arbitrary code via overwriting the function return address (CVE-2002-1904 [28]). This vulnerability can be used to launch a privilege escalation attack against non-control I/O data [9]. Code 5 shows the related code snippet of this attack. The `ptr` variable on line 2 is a pointer that points to the text string of the URL request from clients. This pointer is further passed to the `Log()` function on line 7. At the beginning of the `Log()` function, the register that holds the `ptr` variable is pushed on the stack, which can be overwritten by the memory error on line 15. Therefore, after `Log()` returns on line 7, the `ptr` variable can be corrupted by an attacker and further processed on line 9.

This attack is difficult to detect by existing mechanisms, because all control flows of this program are legitimate. However, SECDINT is able to identify the `ptr` as a sensitive variable, because it is used as a conditional branching data as well as a parameter in the `strstr()` library function. Although the memory error may overwrite the stack memory without other protections, the value of `ptr` is loaded from the enclave instead of the ordinary memory, which prevents this attack from being effective. After enforcing SECDINT on the

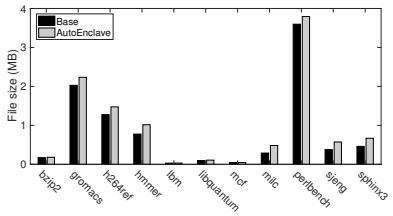

Fig. 7. SPEC space overhead.

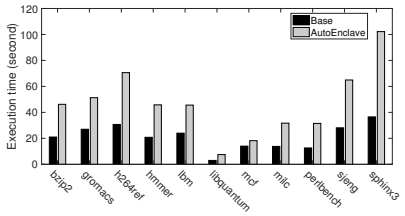

Fig. 8. SPEC execution time overhead.

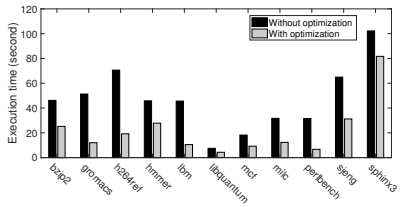

Fig. 9. Performance improvement over optimization.

GHTTPD programs, 48 sensitive variables are automatically identified without the optimization method, and 34 sensitive variables are extracted by the optimization method. The averge execution time overhead is 68.4% without the optimization method and 27.2% with the optimization method.

```
1  int serveconnection(int sockfd){
2    char *ptr;
3    ...
4    if(strstr(ptr,"/.."))
5      ... //reject the request
6    getpeername(...);
7    Log("Connection from %s,request = GET %s",
       inet_ntoa(sa.sin_addr),ptr);
8    if(strstr(ptr,"cgi-bin"))
9      ... //handle cgi request
10 }
11 void Log(char *format,...){
12   char temp[200],temp2[200],logfilename[255];
13   ...
14   vsprintf(temp,format,ap); //buffer overflow}
```

Code 5. Code snippet of GHTTPD.

2)***WU-FTPD***: WU-FTPD is a popular FTP server for Unix-like systems. As mentioned in Section III-A, by making use of a format string error, an attacker can tamper the `pw->pw_uid` parameter associated with the `setuid` system call to escalate the user privilege. SECDINT successfully prevent this attack by isolating the `pw->pw_uid` parameter in the enclave memory. After applying SECDINT on the WU-FTPD applications, 368 sensitive variables are automatically identified without the optimization method, and 134 sensitive variables are extracted with the optimization method. The average execution time overhead is 58.2% without the optimization method and 24.9% with the optimization method.

3) ***Null HTTPd***: Null HTTPd is a multi-threaded web server for Linux. This server has a heap overflow that an attacker can pass a negative content length of value to the server to modify the allocation size of the buffer in memory through the `free()` function (CVE-2002-1496 [4]). In [9], this vulnerability is further utilized to launch a data-oriented attack. The key idea of this attack is to corrupt the CGI-BIN configuration string, which holds the program's directory that could be executed in the future while processing HTTP requests. Since the CGI-BIN configuration string is an I/O parameter in the program, SECDINT identifies it as a sensitive variable and enforces the binary to access it in the enclave. After applying SECDINT on this program, the performance overhead with and without the optimization method are 62%, and 29%, respectively.

E. Discussions and Limitations

Our approach leverages multiple tools to achieve its goals and it may have potential limitations: (1) Our approach integrates UROBOROS [32] as an instrumentation tool to insert SGX instructions into binaries. One drawback of this technique is its inability to fully support C++ disassembly due to the complexity of section structures in binaries compiled from C++. A solution in [32] addresses this issue by dumping and symbolizing the `.ctors` and `.init_array` section. (2) Variables might be referenced by pointers, which are powerful programming constructs. Currently, SECDINT has limitations in identifying variable pointers. We plan to implement point-to analysis to identify the set of memory locations that a pointer may point to. (3) The DFG recovery tool utilized in our approach is based upon the CFG generated by ANGR, employing forced execution, and symbolic execution. These two techniques are effective in managing indirect jumps. Nevertheless, the DFG resulting from these techniques lacks soundness, as it solely mirrors states within the CFG, encompassing the array of potential values. (4) Currently, SECDINT cannot fully identify dynamically loaded library function calls. Because these libraries wait to load until they are needed during execution, statically identify them become challenging.

VII. RELATED WORK

Data-oriented attacks were first demonstrated by Pincus and Baker [27]. They categorized attacks that target only critical data structures as pure data attacks. Then, attacks of this type were extensively studied by Chen *et al.* [9], who showed that these attacks can have serious implications among real-world applications. Hu *et al.* [16] developed a solution to automatically construct non-control data exploits. The key idea is to automatically stitch together multiple existing data flows to corrupt target variables in a program, thereby alleviating the efforts of human analysis. Recently, Hu *et al.* published a more advanced work that can craft data-oriented attacks with rich expressiveness by identifying data-oriented gadgets in programs [17]. Remarkably, the exploited program could still follow the CFG, which makes the attack difficult to combat.

Many mechanisms have been proposed to combat data-oriented attacks. Data-flow integrity ensures that the flow of data in a vulnerable program remains within a data-flow graph [6]. The data-flow graph is generated through static analysis,

making the defense coarse-grained. Another category of defense mechanisms focused on enforcing type- and memory-safety on systems. Softbound [21] has brought memory safety to un-safe C language by using bound checking with fat-pointer. CETS [22] enhanced this method by preventing memory management errors. Importantly, these strategies either suffer from a high performance overhead without hardware acceleration or can only be deployed in very limited usage scenarios [10]. Moreover, these strategies may be got around by advanced attacks due to the inherent vulnerabilities of software-based mechanisms [10].

Another drawback of these defenses is that they require significant changes of a program's source code to adopt the safe dialects. Newsome *et al.* proposed a defense mechanism that does not require source code to protect programs against data-oriented attacks [24]. This approach uses dynamic taint analysis to label all critical data, tracking the propagation of tainted data, and thereby detecting *over-write attacks* by identifying the integrity of tainted data. However, programs protected by this technique run 24 times slower, which proves excessively expensive for widespread adoption.

VIII. Conclusions

In this paper, we presented SecDINT, an automated and practical system harnessing the Intel SGX to protect sensitive variables from data-oriented attacks. We developed an identification method to automatically identify variables that are vulnerable to data-oriented attacks within binaries. Additionally, we designed an optimization approach to minimize performance overhead. Through performing extensive experimental evaluations using Coreutils, SPEC CPU, and multiple real-world applications, our finding underscore SecDINT's effectiveness in accurately identifing sensitive variables, reinforcing data integrity, and thwarting data-oriented attacks. Comparative analysis with existing software-based strategies (e.g., 103% run-time overhead in Data-flow Integrity, 116% in SoftBound with CETS), showcased SecDINT's remarkable capability in drastically reducing overhead to as low as 20.1%.

IX. Acknowledgement

Dakun Shen was supported by Zhejiang Science and Technology Department under Grant No. 2023C01001.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. of ACM CCS*, 2005.

[2] S. V. Acker, N. Nikiforakis, P. Philippaerts, Y. Younan, and F. Piessens. ValueGuard: Protection of native applications against data-only buffer overflows. In *Proc. of ICISS*, 2010.

[3] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE, 2019.

[4] D. Cahill. Heap-based buffer overflow in Null HTTPd Server 0.5.0 and earlier. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1496.

[5] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. of USENIX Security*, pages 161–176, 2015.

[6] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proc. of USENIX OSDI*, 2006.

[7] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 10(6):368–379, 2013.

[8] Quan Chen, Ahmed M Azab, Guruprasad Ganesh, and Peng Ning. Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 167–178, 2017.

[9] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proc. of USENIX Security*, 2005.

[10] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–36, 2021.

[11] Standard Performance Evaluation Corporation. SPEC CPU 2006. https://www.spec.org/cpu2006/.

[12] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee. Efficient protection of path-sensitive control security. In *Proc. of USENIX Security*, 2017.

[13] A. Fog. Calling conventions for different c++ compilers and operating systems, 2017.

[14] A. Fog. Optimizing subroutines in assembly language, 2018.

[15] Tao Hou, Shengping Bi, Mingkui Wei, Tao Wang, Zhuo Lu, and Yao Liu. When third-party javascript meets cache: Explosively amplifying security risks on the internet. In *2022 IEEE Conference on Communications and Network Security (CNS)*, pages 290–298. IEEE, 2022.

[16] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *Proc. of USENIX Security*, 2015.

[17] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proc. of IEEE S&P*, pages 969–986, May 2016.

[18] Intel. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.

[19] O. Levi. Pin - a dynamic binary instrumentation tool, 2018.

[20] Microsoft. Data Execution Prevention (DEP). https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx.

[21] S. Nagarakatte, J. Zhao, M. M.K. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proc. of ACM PLDI*, 2009.

[22] S. Nagarakatte, J. Zhao, M. M.K. Martin, and S. Zdancewic. CETS: Compiler enforced temporal safety for C. In *Proc. of ISMM*, 2010.

[23] J. Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[24] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[25] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N Asokan, and Ahmad-Reza Sadeghi. Hardscope: Hardening embedded systems against data-oriented attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[26] PAXTEAM. PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[27] J. Pincus and B. Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security & Privacy*, 2(4):20–27, July 2004.

[28] qitest1. Buffer overflow in the Log function in util.c in GazTek ghttpd 1.4. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1904.

[29] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):699–742, 2014.

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *Proc. of IEEE S&P*, pages 138–157, 2016.

[31] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[32] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *Proc. of USENIX Security*, 2015.

[33] M. Zalewski. SSH CRC-32 Compensation Attack Detector Vulnerability. https://www.securityfocus.com/bid/2347.